To appear in the Proceedings of International Conference on Distributed Computing Systems, April 2000

Active Files: A Mechanism for Integrating Legacy Applications into Distributed Systems*

Partha Dasgupta
Department of Computer Science
Arizona State University
partha@asu.edu

Ayal Itzkovitz and Vijay Karamcheti
Department of Computer Science
New York University
{*ayali, vijayk*}*@cs.nyu.edu*

Abstract

Despite increasingly distributed internet information sources with diverse storage formats and access-control constraints, most of the end applications (e.g., filters and media players) that view and manipulate data from these sources operate against a traditional file-based interface. These legacy applications need to be rewritten to access remote sources, or need to rely upon ad hoc intermediary applications that aggregate the data into a passive file before executing the legacy application.

This paper presents a simple, elegant, programmable method for allowing natural integration of legacy applications into distributed system infrastructures. The approach called active files, enables multiple information sources to be encapsulated as a local file that serves as their logical proxyw. This local file is accessed though a sentinel process, which automatically starts when the file is opened, aggregates data from multiple sources, and filters all access to and from the file. More importantly, the integration of active files into client applications is transparent: an active file is virtually indistinguishable from a regular file. Active files find a variety of applications in both distributed and non-distributed systems. We discuss active files, their semantics, their usage and their implementations in Windows NT.

1. Introduction

The growth of the internet has been accompanied by an explosion in information sources. These sources are distributed, store data in different formats, are dynamically changing, and may be subject to a variety of constraints including consistency, privacy, and access-control. However, most of the end applications that view and manipulate data from these sources (e.g., browser helper applications such as filters and media players) assume a traditional file-based interface, treating files simply as a passive, persistent, uninterpreted sequence of bytes. Consequently, the integration of such legacy applications into distributed systems has required either significant code modification to use a new distributed system-aware API or relies on the ad hoc use of intermediary applications that isolate the end application from the data sources. These intermediaries perform necessary operations such as access control, filtering, and format conversion before aggregating the data into a passive file that can be handed down to legacy applications.

However, there are significant shortcomings to both these approaches. The first, supported by constructs such as dynamic HTML and cgi-bin scripts to couple activity with network-accessed files, and component frameworks such as DCOM, and CORBA to support object-specific access interfaces, has seen restricted use except in a few scenarios. Reasons include the vastly different semantics provided by these constructs, their relatively complicated APIs, and their heavyweight implementations. The second approach although more popular, has the disadvantage that the data collected by the intermediary is completely decoupled from both the original sources of the information and the end application. Consequently, it is unable to track changes in the original sources or be controlled by the end application. For example, an end application that searches through a collection of distributed databases cannot see changes in these databases, nor influence the progress of the search when an intermediary first aggregates data from these databases and presents it to the search application as a file.

In this paper, we present an elegant, easy to use, and programmable concept, that allows natural integration of legacy applications into distributed system infrastructures. This association is done by a construct we call *active files*, which enables distributed sources of information to be encapsulated in the form of a local file that serves as their logical proxy. An active file has the look and feel of a regular file, but is associated with a sentinel (process) that can

*This research was sponsored by DARPA/AFRL-Rome agreements F30602-96-1-0320 and F30602-99-1-0517, by NSF award CCR-9876128, and Microsoft.

act on the streams of data that enter the file on a write or exit the file on a read. More importantly, from the perspective of the end-application, active files are indistinguishable from non-active files. There is no reprogramming, or recompilation necessary for using active files. The sentinel can perform a variety of tasks, including aggregating data from distributed sources, filtering data entering/exiting the file, and performing actions that have external side-effects.

Active files provide a convenient abstraction for alleviating several shortcomings of intermediary approaches. The sentinel process can control flow of data between distributed sources and the end application, enforcing consistency, privacy, and access-control constraints required by the former while simultaneously yielding control to the end application. For example, an active file can provide the illusion of accessing a single file even though the file data is physically located on multiple remote sites with varied authentication and access-control policies. In addition, it can monitor how the application uses this file, caching only the most frequently accessed contents for performance. Moreover, the cache can be kept consistent with any updates performed to its contents at any of the remote sources. Note that all of these behaviors can be expressed without modifying either the end application or the original sources of data.

Active files can also enhance regular file functionality in non-distributed systems. Traditionally, once a regular file is made accessible to a process, no control can be exercised on when or how the process uses the file. In general, the owner/creator of a file may wish to control and log its accesses, filter the data supplied or stored, or may just want some side effect (such as notification) to be triggered as a result of the access. For instance, a logfile that accepts log entries from many processes may want to enforce some form of locking. A file containing sensitive data would like to log every access from users, even if these users are trusted users. Active files provide an elegant mechanism for expressing many such diverse applications.

Active files differ significantly both from approaches such as Ufo [1] and Prospero [13] that overload/extend the file system interface to provide seamless access to remote files, and approaches such as Watchdogs [3] that rely on kernel support for notification about file access. Unlike the hard-coded functionality of the former, active files are completely programmable, enabling the expression of general per-file behaviors in a simple, uniform, and conceptually elegant fashion. Moreover, even though an access notification mechanism is sufficient to implement locking, filtering, and other features, the heavyweight nature of kernel involvement restricts its applicability. In contrast, active files can be implemented efficiently at the user level, and consequently can be used for a much larger set of applications.

We describe an implementation of active files in Windows NT. Our implementation relies on the binary interception of Win32 file API calls [2, 11]. The intercepted calls enable the sentinel process to both attend to application demands and constraints of the distributed information sources, without requiring the active participation of either. We show that several implementation approaches are possible that trade off cost for programming convenience.

The remainder of the paper is as follows. Sections 2 and 3 discuss semantics and uses of active files, with their implementation and programming described in Sections 4 and Section 5. The performance overheads of active files are presented in Section 6. Section 7 discusses related work. Appendix A describes the Windows NT implementations.

2. Active Files

An active file is a regular file that is associated with an executable program. When an active file is opened, the associated executable is run as a *sentinel* process. The sentinel connects with the user process using pipes and can directly access both the remote information source(s) and the local file. User process writes are sent to the sentinel along the write pipe, and user process reads extract data out of the read pipe. Logically, the sentinel contains two threads that handle the flow in both directions between the user process, the remote sources, and the local data file (see Figure 1).

2.1. File system interface

An active file is represented in the file system by two passive files: a data file, and an executable. Directory operations such as creating, copying, and deleting result in corresponding operations on the passive components. For instance, a copy operation produces a second active file with the same data and executable components as the first one.

User processes interact with a sentinel process using standard file API calls such as `CreateFile`, `OpenFile`, `ReadFile`, `WriteFile`, and `CloseHandle`. Other API calls such as `GetFileSize` are passed on to the sentinel for handling as appropriate. Consequently, from the user process' perspective, interactions with active files are indistinguishable from interactions with ordinary (passive) files. Associating the file handle used by the user process to the two pipes is the responsibility of the implementation.

2.2. Semantics of the sentinel process

The sentinel process is started and terminated when a user process opens and closes the active file. If multiple user processes open the same active file, multiple sentinels are created, which synchronize amongst themselves in a program-dependent fashion using semaphores, shared memory or other forms of interprocess communication (IPC).

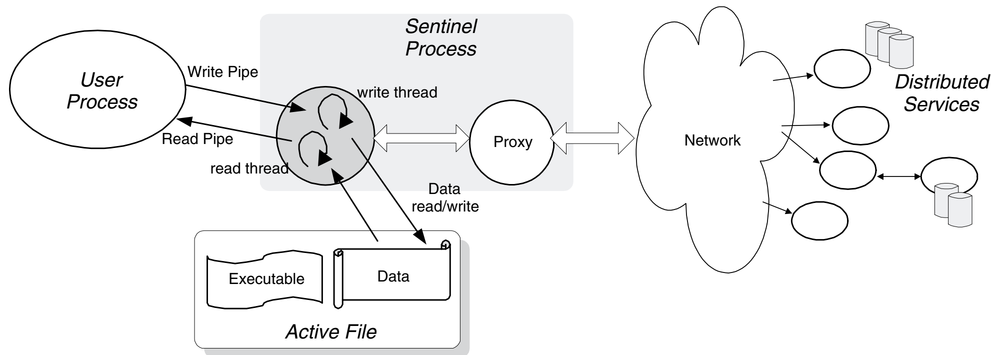


Figure 1. The logical view of an active file and a user process.

The sentinel process is best viewed as an entity that aggregates information from and distributes information to remote sources, serving as a two-way filter between the user application and the information sources. The data file associated with an active file acts as a local cache. The active file processes data sent to it by the user process, writing it to the data part and optionally also sending it to the remote location. It also reads the data part of the active file, processing it before making it available to the user process. The sentinel can be a null filter, in which case the active file has the semantics of a passive file. The sentinel can also be practically any program; the system puts no restrictions of its capabilities. Note that an active file can have an empty data part. In this case, the sentinel either directly interacts with the user process by producing/consuming required data, or acts as a conduit between the application and the network.

Writing the sentinel process is straightforward, although the specifics depend on how the abstraction is implemented. Figure 2 shows the code for a null filter in the simplest implementation strategy, which directly mimics the logical abstraction. The sentinel process consists of two threads. The first reads data in from the network, writes it to the data file (the cache) and then sends the data to a read pipe that transports it to the application. The second thread reads off of the write pipe and writes the received data to the cache as well as forwards it on to the original source. Section 4 reports on other implementation strategies and associated sentinel programming. These strategies support handshaking between the user and sentinel processes, and represent more aggressive implementations where some functionality is migrated between the sentinel and user processes.

2.3. Security implications

Although this paper does not focus on the security aspects of active files, a few points need discussion. Opening an active file is predicated upon access to the passive file components, and launches a program under the user-id of the application that opened the file. This program can, of course have any side effect, including malicious ones, such as destroying data and activating viruses. However, these effects are no different from those initiated by any other executable started under the same user-id. Note that the operating system already places restrictions on how the latter can access the user machine. In applications with additional security requirements, orthogonal techniques such as certificates, code signing, and sandboxing [9] can be used.

3. Uses of Active Files

The active file is a general-purpose construct that has a large number of potential applications in both sequential and distributed systems, limited only by what can be expressed in the sentinel process. Similar to concepts such as files, pipes, and scripts, the active file can be used for many scenarios, when combined with other system programs.

In general, the sentinel process can encapsulate four fundamental actions (see Figure 3 shows these actions): (1) *data generation*, (2) *input* and *output filtering,* (3) *aggregation*, and (4) *distribution*. The first two primarily interact with the local data file, while the other two involve remote information sources. Larger applications are constructed by composing these actions in different ways. Note that the data file need not actually exist: the sentinel process just creates the illusion of its existence for client applications.

Data generation The sentinel process can completely obviate the existence of a physical (passive) file that stores the data associated with the active file. An example of such use is when the sentinel process just contains a random number generator. In this case, the corresponding active file appears to client programs as a data file that contains an infinite stream of random numbers.

Input and output filtering The sentinel can introduce actions on either all or a subset of the read and write accesses to the active file. This admits a range of uses, from keep-

```
HANDLE hin, hout, hcache, hpipe;

DWORD RWThrd(DWORD dir) {
  char  buf[1024];
  DWORD rbytes, wbytes;

  while( ... ) {
    if (dir == READ) {
      /* read from remote source */
      ReadFile(hpipe,buf,1024,&rbytes,NULL);
      WriteFile(hout,buf,rbytes,&wbytes,NULL);
      WriteFile(hcache,buf,rbytes,&wbytes,NULL);
    } else {
      /* write to remote source */
      ReadFile(hin,buf,1024,&wbytes,NULL);
      WriteFile(hcache,buf,wbytes,&rbytes,NULL);
      WriteFile(hpipe,buf,wbytes,&rbytes,NULL);
    }
  }
  return 0;
}

int main(int argc, void *argv[]) {
  HANDLE hthrd[2];
  DWORD  tid;

  /* create handles */
  hin = GetStdHandle(STD_INPUT_HANDLE);
  hout = GetStdHandle(STD_OUTPUT_HANDLE);
  /* handles to source, cache */
  hpipe = OpenPipe(argv[1],...,...);
  hcache = OpenFile(argv[2],...,...);
  /* create threads */
  hthrd[0]= CreateThread(0,0,RWThread,0,0,&tid);
  hthrd[1]= CreateThread(0,0,RWThread,1,0,&tid);
  WaitForMultipleObjects(2,hthrd,TRUE,INFINITE);
}
```

Figure 2. Sentinel implementing a null filter.

ing a log of actions to *filtering* the data read from and written into the data file. A simple example of such filtering is a compressed file. In this case, the sentinel process compresses and decompresses the file data as it is written and read. An advantage of this approach over compressed file systems is that file compression can be handled on a per-file basis with different compression algorithms used for different types of files. Additionally, both compression and decompression can be demand-driven and performed incrementally. Note that the client application is completely unaware that it is interacting with a compressed file.

Filtering can also be used to provide a file-based interface to the Windows system registry, considerably simplifying system configuration. The sentinel checks the registry, providing a simplified version (e.g., a plain text file) to the client application. Any modifications by the client application can in turn be parsed by the sentinel process and translated into appropriate registry modifications.

An active file can also combine logging and filtering actions for concurrent and intelligent logging. Assume that several processes log events using the same log file. As the

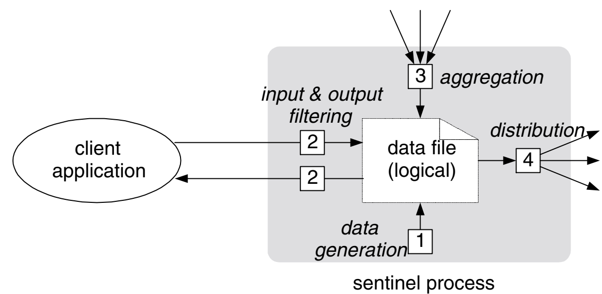


Figure 3. Fundamental active file actions.

sentinel receives each log record, it locks the file, writes the record and unlocks the file. The processes generating the logs do not need to know about log file locking. Moreover, the sentinel can perform a variety of functions in the background such as cleaning up the logs. Achieving similar functionality with passive files would require the client applications to essentially embed all of the code and locking protocols for the log managers.

Aggregation The sentinel can aggregate information from various sources, presenting it to client applications as a conventional file. Examples of these sources include other local or remote files, databases, network connections, or even other processes.

An example of active-file based aggregation is seamless access to remote files that are not accessible via network-mapped shares. The sentinel accesses the remote file using a standard protocol (e.g., FTP or HTTP), creates a local copy, and makes the copy available to the client application. The sentinel can also merge multiple remote files into a single local file. From the client's viewpoint, remote file accesses are indistinguishable from local ones. Similar transparent access to remote files can also be provided without ever making a local copy. The sentinel directly reads data from and writes data to a network connection.

Aggregation can also be used to dynamically construct files containing data from various sources. An example might be an active file that reflects the latest stock quotes (downloaded by the sentinel from a server) every time the file is opened. Similarly, an inbox file of an E-mail program can be such that reading it causes new messages to be retrieved possibly from multiple remote POP servers.

Distribution Sentinel processes can also distribute information to various sources, triggered by file operations against the active file. As with aggregation, these source include other local or remote files, databases, network connections, and other processes.

An example of active-file based distribution is an E-mail application where the outbox-file can be programmed to send email to a particular recipient, every time some data is written to it. This concept can be extended such that the

sentinel process parses the data written to the file to extract the "To" addresses and send the data to each recipient.

In general, the sentinel process can be used to produce side effects in the active file's environment. These side effects can be both synchronous (i.e., triggered by file operations) or asynchronous (i.e., take place in the background).

4. Implementing Active Files

We present four approaches to implement active files. Common to all the approaches is the ability to intercept system calls (and especially those for file system). While this is a general technique known to work for many operating systems, we describe our implementation for Windows NT.

Using interception, we redivert, at runtime, the file system API calls initially intended for the Kernel32 DLL, to stub functions that implement the features of the active files. We use the "*Mediating Connectors*" toolkit from USC/ISI [2] to perform this interception. The toolkit allows simple runtime interception and replacement of selected Win32 API calls (or calls to any DLL) with calls to another routine. Moreover, interception can be done in a secure fashion such that the application cannot undo it.

The four approaches—*process, process-plus-control, DLL-with-thread, and DLL-only*—reflect different partitioning of the functionality of active files between the user and an external process, and represent different tradeoffs between runtime overheads and the convenience of programming active files. We sketch the implementation approaches below, deferring a discussion of how the sentinel is programmed in each case to Section 5. Appendix A provides Windows-NT specific implementation details.

4.1. Process-based implementation

The process-based implementation approach is the simple and intuitive method, directly reflecting active file semantics. When a process performs an OpenFile (or CreateFile) operation on an active file, the call goes to a stub routine, which first creates a new process for running the executable associated with the active file. The stub also passes the created process the name of the data part of the active file for use by the sentinel. Then, the stub creates two pipes and attaches them to the standard input and output of the sentinel process. Finally, the stub stores the handles of the pipes in a structure, returning to the application process a fictitious handle that points to this structure.

The ReadFile/WriteFile calls are also instrumented. The instrumented calls translate operations on the fictitious handle into reads or writes on the appropriate pipe.

This straightforward implementation approach though convenient to program to, suffers from two problems. First, it can only support a subset of the file operations. Operations such as ReadFileScatter (or *seek* in Unix) and GetFileSize cannot be implemented as there is no method of passing control information between the user process and the sentinel process. The second problem is performance: each file operation requires two protection-domain crossings. These shortcomings are alleviated in the rest of the implementation approaches, albeit at the cost of slightly increasing the difficulty of programming active files.

4.2. Process-plus-control based implementation

This approach solves the problem of handshaking between the user and sentinel processes by adding a control channel in addition to the two pipes. As before, the active file stub functions handle all interactions with the control channel and the data pipes. The user process continues to interact with an active file using standard file operations.

In this approach, all API requests from the application are first transmitted to the sentinel process via the control channel and the response of the sentinel process is read from the read pipe. So when the application process wants to read 50 bytes, a "*read 50"* command is sent to the sentinel, and then 50 bytes are read from the read pipe. When the application wants to write 30 bytes, a "*write 30"* command is sent on the control channel and then 30 bytes are written to the write pipe, which is retrieved by the sentinel process.

Hence, all other file operations are now passed to the sentinel process as commands with arguments. The active file stubs read the results off the read pipe, and present appropriately packaged responses (as return codes, or structures) to the user process. A set of headers is provided to the application programmer to enable easier handling of the control channel when writing the sentinel process. Details of the programming interface are discussed in Section 5. Note that the writer of the sentinel process has complete flexibility in deciding when to listen to the control channel. Given different models of usage, the sentinel process might choose to eagerly inject data into the read pipe (anticipating read requests from the user) and eagerly wait to read data from the write pipe (anticipating write requests).

Both the process and process-plus-control implementations of active files are clean and useful. However, as expected, and as verified later in Section 6, they suffer from performance problems, being relatively heavyweight due to excessive context switching and data copying. The additional functionality provided by active files can offset these performance problems in most cases. However, when performance is an overriding concern, active files can be made more efficient by trading off some of their programming convenience and trespassing into the protection boundaries of the application process. We describe two such approaches in the remainder of the section.

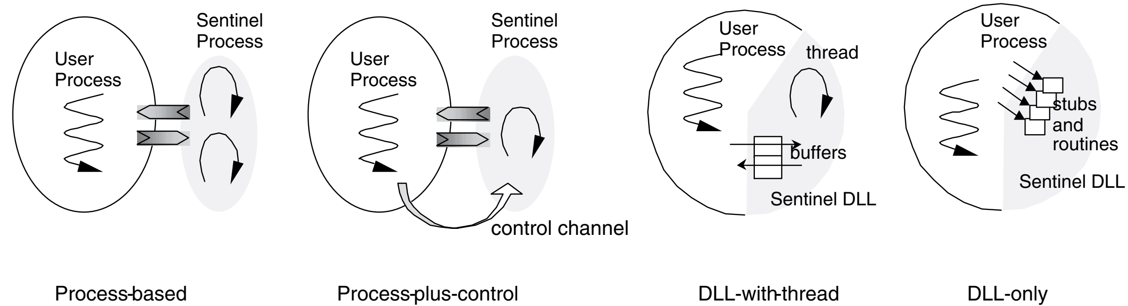


Figure 4. Four implementation approaches.

4.3. DLL-with-thread based implementation

Instead of a stand-alone process, this approach encapsulates sentinel fnctionality into a separate DLL, referred to as the sentinel DLL. The initialization routine of this DLL, activated at load time, is the one responsible for orchestrating interactions between the user application, the remote information services, and the local data part of the active file.

Opening an active file "injects" [15] the sentinel DLL associated with the file into the application and starts a thread for running the orchestration routine. An application write results in the write stub signalling a write to the sentinel thread. The sentinel thread then provides a buffer to the active file write routine, which copies data from the user buffer to this target buffer. Reads are handled similarly. While the implementation preserves active file interface and semantics, its architecture is somewhat non-intuitive.

The sentinel process is no longer a process running separate from the application, but just a thread in the application. There is no inter-process context switching needed – enhancing the performance. File data is not copied from user space to kernel space and then to user space (as is the case with pipes), instead using only one user-level copy.

4.4. DLL-only based implementation

Although the DLL-with-thread approach performs significantly better than the other two (see Section 6), it still has a performance limitation: all file operations require context switching between the requesting user thread and the sentinel thread. The DLL-only implementation approach eliminates this switch by directly routing file system API calls to appropriate routines in the sentinel DLL.

The active file programmer can express any functionality in these routines including additional thread and process creation. Although this approach provides the same functionality as the other approaches, the programmer needs to be involved in the low-level details of writing the sentinel DLL and cannot take advantage of a clean simple interface.

5. Programming the Sentinel Process

We describe below the programming of the sentinel process for each of the above approaches. As expected, the process-based implementations present active file implementers with a simpler interface than do the DLL-based implementations. One can define a common API that spans across all implementations; however, this complicates the simple programming in the first two cases. We are currently exploring automatic translation strategies for taking an active file written for a process-based implementation and producing the DLLs necessary in the DLL-based strategies.

5.1. Process-based implementation

Figure 1, described earlier, showed how a sentinel process is written in the process-based active file implementation. This case uses two threads, one that collects data from information resources and/or the data portion of the active file and makes it available to standard output, and another thread that reads in data from standard input and distributes it to the information sources, optionally updating the local cache part. This approach has the advantage that the sentinel process can be developed as a standalone executable independent of its interactions with other processes.

5.2. Process-plus-control based implementation

The sentinel process in this implementation involves only a single thread, which typically blocks on a read on the control channel. Upon receiving a command from the application, the thread wakes up and performs the operation (which might entail putting data into the read pipe or taking data off the write pipe). This implementation relies on a description of all the control messages that can be expected on the control channel. Return codes, if any, are passed back along with the data via the read pipe. Again, the formats of such return values must match the formats expected by the application stubs and are defined by our implementation.

5.3. DLL-with-thread based implementation

In this implementation, the sentinel process is no longer a process but a thread, started in a routine called SentinelThrdMain that has to be defined in the DLL associated with the active file. The thread can use three library calls to communicate with the application. The three calls are:

1. AF_GetControl – gets control messages sent by the application to read/write data or perform other file operations. These messages might affect the information sources or only make local state changes.
2. AF_SendDataToAppl – communicates read responses or file information requests to the application.
3. AF_GetDataFromAppl – gets client writes.

Similar to the process-plus-control strategy, the thread in the SentinelThrdMain routine runs a dispatch loop using calls to AF_GetControl.

5.4. DLL-only based implementation

The programming interface in this case is simply a pass-through, which passes the API calls generated by the application unmodified to the sentinel. The sentinel is a DLL that replaces Win32 file system calls, with calls programmed by the active file implementor. This DLL has to provide a set of routines such as OpenFile, CreateFile, ReadFile, and WriteFile. These routines are invoked whenever the application accessing the file calls the corresponding Win32 functions. This clearly is the most efficient implementation, however places the most burden on the programmer.

6. Overhead of Active Files

This section present the performance of three different implementation of active files, in which the sentinel is a process in the local machine, an injected thread, or a direct DLL implementation of the Read/Write file operations.

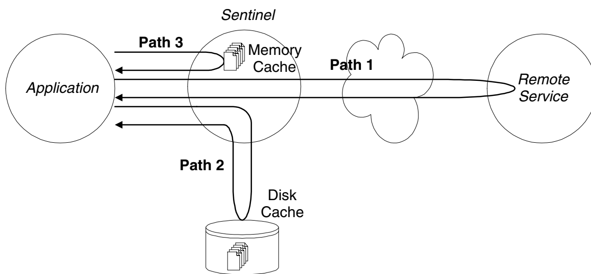


Figure 5. Three critical execution paths.

Figure 5 shows three different critical paths of execution of an active file, emulating different caching options, carried out by the sentinel. **Path 1** represents the case of no cache in the sentinel process. Whenever the application issues a Read, a message is sent on the control channel, asking the sentinel to retrieve the data from the remote service. When the data arrives at the sentinel, it pushes it over to the application to satisfy the Read operation. When an application issues a Write, the buffer is sent directly to the sentinel, which then sends an update message to the remote service.[1]

Path 2 represents the case where the data is cached in the active file on disk. Here, the sentinel interacts with its local file rather than contacting the remote service for getting or updating data, driven by application needs.

Path 3 represents a similar case as the second path, except that the cache resides in the sentinel's memory rather than on disk. The sentinel code uses a memory buffer to store application Writes or respond to application Reads.

Performance Results We ran the experiments on a cluster of PCs (300 MHz Pentium II), interconnected with 100Mbps Fast Ethernet. Figure 6 shows measurements for an application that reads and writes fixed-size blocks from an active file (we instrumented the application by intercepting the open/read/write/close calls and handling them as described before). Our measurements are for a variety of block sizes, and time 1000 calls of each.

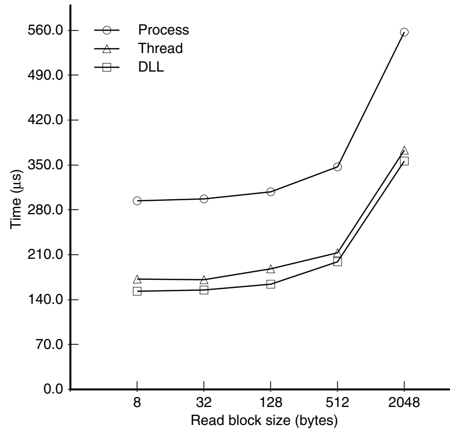
The Read and Write results reflect the latency and bandwidth effects of the sentinel respectively. Since an application is blocked on a read operation, the overhead of interaction with the sentinel process and any processing therein adds to the latency of the read operation. Since writes are issued without waiting for their completion, any increase in the overhead of a write stems from bandwidth restrictions imposed by the sentinel interaction and processing.

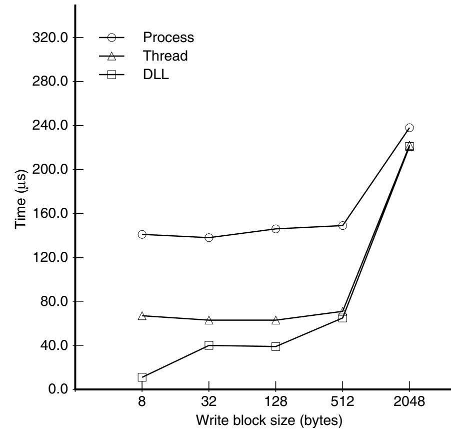
As the graphs show, the different active file implementations impact the latency and bandwidth of file operations to different extents. The process-based implementation has the largest impact on latency and bandwidth, while the DLL-only implementation has negligible impact incurring the same costs as if the application were directly accessing the information sources. The thread implementation represents an intermediate point between these extremes, trading off performance for programming convenience. The DLL implementation introduces only a very thin layer of code (injected into the application process); consequently, it incurs no extra system calls or context switches.[2]

The process implementation's overheads stem from the extra buffer copying and process context switching occur-

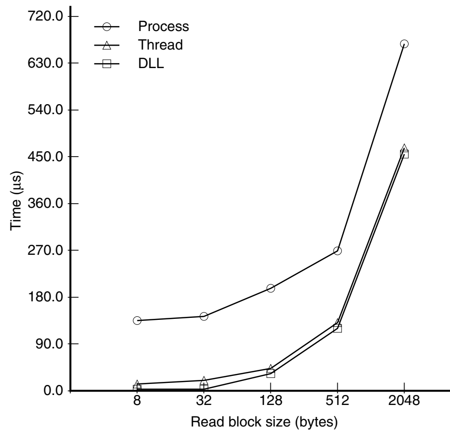
[1] For clarity, we only describe the unoptimized interaction between the application and the sentinel. The implementations are optimized to improve buffer reuse and reduce synchronization overheads.

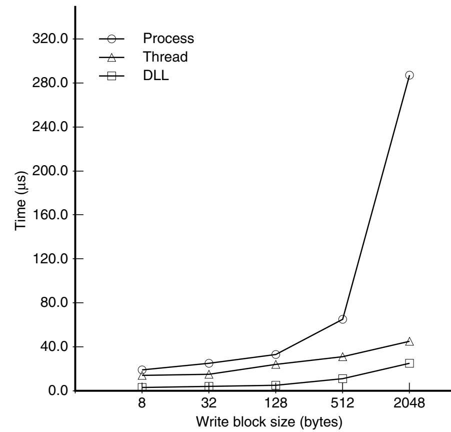
[2] The Read operation, normally a system call, is sometimes diverted to a user-mode memcpy() call improving performance over the original.



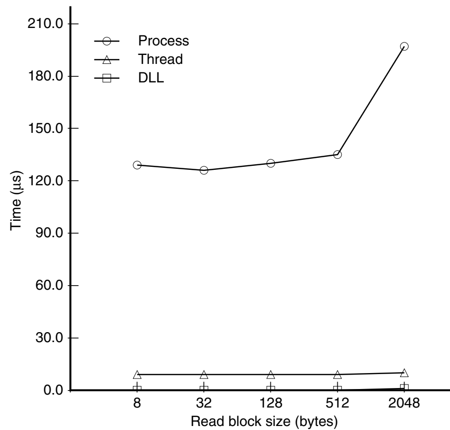


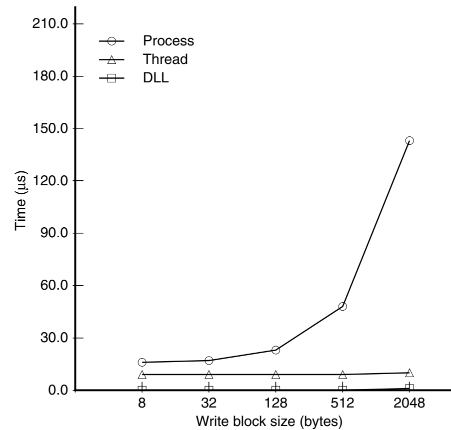
(a) Sentinel uses a remote source.





(b) Sentinel uses a local on-disk cache.





(c) Sentinel uses an in-memory cache.

Figure 6. ReadFile and WriteFile overheads (in μs) of different active file implementations—process-with-control (Process**), DLL-with-thread (**Thread**), and DLL-only (**DLL**)—for three critical caching paths that involve the (a) network, (b) local disk, and (c) local memory respectively. The baseline costs for directly accessing these paths is indistinguishable from the DLL-only case and is not shown.**

ring in the critical path. Completing the read operation requires a thread in the sentinel process to receive the read request, copy the buffer, send a message, and context switch before the application thread can finish the operation. The results for the Write case are a bit better because data streaming hides some of the latency. However, each Write still requires at least one buffer copy and message receive (and two context switches between processes).

The thread implementation, on a file operation, lets the application simply switch over to the sentinel thread, which performs the necessary operations without requiring costly interactions across process boundaries.

Note that the above measurements represent only baseline overheads of interacting with the sentinel in different implementations. Since overheads incurred within the sentinel are influenced by its functionality, it is difficult to predict what these costs might be in a particular case. However, our results above show that the active files framework on its own does not introduce extra cost: the eventual cost of using active files is determined only by the *functionality* that they implement, not by the cost of *interacting* with them.

7. Discussion and Related Work

Active files permit natural integration of legacy applications into distributed systems by automatically interposing a sentinel process between a legacy application written assuming a traditional file-based interface and one or more remote sources of information. Our mechanism can be viewed in the context of two groups of related work: those efforts whose goal is to allow legacy applications to interface with distributed information sources, and those that rely upon kernel support to extend file-system functionality.

Accessing distributed information sources The traditional approach to using legacy end applications in distributed environments has relied upon the ad hoc use of intermediary applications that first aggregate the data from remote sources and filter it, before passing it on to the legacy application. Although suitable for some applications, a shortcoming of this approach is that the aggregated data is decoupled both from the original sources of information and the end application, which may want to control the aggregation process. In contrast, the sentinel process interacts with both the end application and the remote sources, and can both track changes in the original sources as well as be controlled by the demands of the end application.

A more direct approach for accessing remote sources relies upon code modification to use a new distributed system-aware API. These APIs exemplified by component-based approaches such as OLE [12], COM [16], DCOM [8], and CORBA [14], allow remote sources to be accessed through well-defined interfaces: the interface permits actions in addition to just accessing or updating the data. Despite their generality, these frameworks have seen restricted use because of differences in semantics, complicated APIs, and heavyweight implementations. One contribution of the active files mechanism is to encapsulate use of these frameworks within a sentinel process, which can optionally cache some of the accesses while presenting the client with an intuitive and familiar file-based interface. Such use has the potential of effectively addressing two of the shortcomings of component-based approaches: ease of use and efficiency.

In some cases, access to remote sources of information requires server-side processing, exemplified by approaches such as dynamic HTML, cgi-bin scripts, and servlets [5]. Active files can easily emulate such behavior with the sentinel process either creating a remote process on demand, or talking to an existing service. Moreover, active files can support applications not possible with these approaches such as aggregating content from various sites.

Extending file-system functionality The mediating behavior of the sentinel process can be thought of as generalizing existing operating system structures such as named pipes, file-based interfaces to devices (e.g., /dev in Unix), and file-based interfaces for process management (e.g., /proc in Solaris). Active files are similar to these structures in providing a connection-oriented service between client and server processes, but differ in their ability to support the complete file-access API by virtue of the presence of the control channel. The latter feature makes them completely indistinguishable from normal files, facilitating their use in many diverse scenarios. Moreover, instead of requiring that client-server interactions always cross protection domains, Active files admit a spectrum of implementations that tradeoff functionality versus runtime overhead.

Several approaches for extending file-system functionality have been proposed. Most of these approaches rely on kernel modifications that enable the stacking of vnodes and templates as in the SunOS [17], Alex [4], and Ficus [10] file systems, or notification about file access as in the Watchdogs [3] approach. Extensible operating systems such as SPIN [7], Exokernel [6], and VINO [18] rely on more aggressive modifications to permit introduction of user-specified kernel functionality. Although these mechanisms provide the necessary hooks to implement active file-like functionality, they are either too heavyweight or are inapplicable on commodity operating systems. In contrast, active files can be implemented efficiently at the user level, permitting their use for a much larger set of applications.

Some recent efforts such as Janus [9] and Ufo [1] have used API or system-call interception to extend file-system functionality without requiring kernel modification. Janus restricts the set of files a process can access and Ufo provides seamless access to remote files. In contrast to the hard-coded functionality of these approaches, active files are completely programmable, enabling the expression of

these and other general per-file behaviors in a simple and uniform fashion. Moreover, unlike both these systems that implement process-centric control, active files enable resource-centric control: the file itself can specify the kind of access control policies that need be implemented as well as custom aggregation and caching behaviors.

8. Conclusions

Active files are a new extensible and programmable concept for transparently associating actions with local data files. Because active files use a familiar file interface for reading and writing data and can be easily programmed with user-specified functionality, they enable the natural integration of legacy applications into distributed environments by making it possible for such applications to seamlessly access remote services. We have described four user-level approaches for implementing active files in the Windows NT operating system. These approaches possess different efficiency and programming characteristics, with the most efficient implementation incurring negligible overheads as compared to direct application-level operations.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, Aug. 1998.

[2] R. Balzer. Mediating connectors 2.0, 1998. `http://www.isi.edu/software-sciences/safe-execution-environments.html`.

[3] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. In *Proc. 1988 Winter USENIX Conference*, pages 267–275, 1988.

[4] V. Cate. Alex - A global filesystem. *Proc. USENIX File Systems Workshop*, pages 1–11, May 1992.

[5] WWW Consortium. `http//www.w3.org`.

[6] D. R. Engler et al. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th Symp. on Operating Systems Principles*, 1995.

[7] B. N. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th Symp. on Operating Systems Principles*, 1995.

[8] R. I. Frank E. *DCOM : Microsoft Distributed Component Object Model*. IDG Books Worldwide, 1997.

[9] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proc. 6th Usenix Security Symposium*, July 1996.

[10] R. G. Guy et al. Implementation of the Ficus replicated file system. In *Proc. 1990 Summer USENIX Conf.*, June 1990.

[11] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. 3rd USENIX Windows NT Symposium*, July 1999.

[12] *Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*. Microsoft Press, 1998.

[13] B. C. Neuman. The Prospero File System: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, Fall 1992.

[14] Object Management Group. `http://www.omg.org`.

[15] J. Richter. *Advanced Windows (3rd Edition)*. Microsoft Press, 1997.

[16] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[17] D. S. H. Rosenthal. Evolving the Vnode interface. In Proc. 1990 Summer USENIX Conference, 1990.

[18] M. Seltzer, Y. Endo, C. Small, and K. Smit. An introduction to the VINO architecture. Technical Report TR-34-94, Harvard University, Cambridge, MA, 1994.

A. Active Files on Windows NT

Active files have an active part and a passive part. Both are saved as (passive) files, relying on NTFS streams capability to package them as a single data file, which exhibits compatible behavior for standard file operations such as copying and renaming. The active part is either an executable (in the process-based approaches) or a DLL (in the DLL-based approaches), while the passive part is a data file. Note that as discussed earlier the data file can be empty.

Whenever an instrumented application tries to open an active file, the active part of the file is executed as a process, or injected as a DLL into the application. To implement this action, we leverage the fact that in Windows NT, the operating system's functionality is found in DLLs. Therefore, Win32 applications are complied with "loose links", with address resolution of API functions done at loading time. At compile time, the linker constructs an import address table (IAT) for the process, which becomes the target for all API calls. At load time, the appropriate DLLs containing the implementation of these APIs are loaded and the addresses in the IAT table are resolved [2, 11, 15].

We manipulate the import table of a running process, so that it can use active files. The idea is that all file operations (that reside in kernel32.dll) can now pass through our injected DLL, which implements the connection between the client application and the active part of the active files; all without the client application being designed (or prepared) for it. We use the Mediating Connectors [2] toolkit to provide the DLL injection and API interception functions [15].

Specifically, the client application, when executing file operations will call the corresponding Win32 API-functions (e.g. OpenFile, ReadFile, WriteFile, etc.). These calls are diverted to active file implementation routines.

A.1. Implementation details

Our implementation consists of two parts, the code to be put in the application and the code to be put in the sentinel. The code that needs to become part of the application is put in a DLL and injected into the application. Note that this code

is part of the active file implementation: it is different from the code contained in the end-user application for using the active file, which accesses the active file as it would an ordinary (passive) file. Moreover, the application-side code is independent of the active file functionality, that is, all active files share this code. The sentinel code also has two parts, a part that is written by the programmer of the active file (the actual logic of the sentinel process) and the code that is supplied by the active file implementation (in the form of headers and library calls.) Again, the headers and library calls are independent of the code written by the programmer. The combination of the programmer-written code and the supplied code is called the active file sentinel code.

A.2. Process-Based Implementations

In both process-based implementations, the application-side code is quite straightforward. It is a set of stubs, one for each instrumented API call. For example, the stub for OpenFile() (or CreateFile) checks to see if the file name corresponds to an active file or not (by checking the extension). If the file is not an active file, the stub calls the standard Win32 OpenFile routine. If the file is an active file, either two (in the simple process based approach) or three (in the process-plus-control based approach) anonymous pipes are created. These pipe handles are duplicated using the DuplicateHandle function. Next, a new process running the executable associated with the active file is started and passed the duplicate pipe handles. Finally, a dummy handle is acquired and supplied as the return file handle to the process that initiated the OpenFile call. An association is also made between the dummy handle and the two or three pipe handles associated with the active file. The CloseHandle call just shuts down the created pipes.

The application-side code for the two process-based implementations differs slightly in how it handles file operations on an open file. In the simple process-based implementation, whenever the application calls ReadFile on some file handle, our stub gets control. The stub checks if this ReadFile is against the dummy handle we created. If not, we pass it to the file system. If yes, then we read the requested amount of bytes from the read pipe using the actual pipe handle. The WriteFile operation is implemented similarly. Operations such as ReadFileScatter that do not have direct correspondence with operations on pipes are simply dropped (with an appropriate return code) by the client stub.

The only difference in the application stub for the process-plus-control based implementation is that it writes the command into the control pipe prior to reading or writing from the data pipes. These commands are picked up by the sentinel process, which either injects data or extracts data as required at the other end. Note that the control channel allows the active file programmer to provide application-specific functionality even for calls that do not have corresponding pipe operations. These calls are just passed on to the sentinel process, and potentially influence data put into the data pipes. This implementation provides the server code with header files containing information about the types of control messages supported on the control pipe, and library stubs to interpret these messages.

A.3. DLL Based Implementations

Instead of having the sentinel run as an external process, the two DLL-based implementations fold the sentinel into the application process. The sentinel code is injected into the process as part of the DLL that in previous implementations contained only the client stubs. The DLL-with-thread based implementation uses library routines to simulate the data transfer and synchronization. There are six routines, three on the client side and three on the server side:

1. AF_SendControl: This routine sends a control message from the client to the server. The routine does not block.
2. AF_GetControl: This is used in the server to get a control message. Blocks till such a message is received. Of course, these “messages” are implemented using events and shared memory.
3. AF_SendDataToSentinel: A client sends data to the server (on writes) using this routine. As before, the data is passed using a shared memory buffer.
4. AF_GetDataFromAppl: Provides complementary functionality to AF_SendDataToSentinel: Blocks until data is received.
5. AF_SendDataToAppl: Similar to (3), but in the other direction.
6. AF_GetDataFromSentinel: Similar to (5), but at the client end.

The OpenFile API call, instead of running the executable as in the process-based implementations, loads the DLL containing the sentinel and starts a thread that runs a routine called SentinelThrdMain. SentinelThrdMain is the replacement for the main() routine used in the earlier approaches. Reading, writing and other control commands call the above library routines as appropriate.

The DLL-only based implementation is actually even simpler. The implementation just substitutes calls to special routines contained in the sentinel DLL (these routines must be named AF_ReadFile, AF_WriteFile, and AF_Control), whenever the corresponding file operations are called by the user program. Note that this approach requires the implementer of the sentinel to handle all of the synchronization and data management.